- The insertion sort is commonly used to sort subarrays smaller than the cutoff.

- The insertion sort can also be applied to the entire array, after it has been sorted down to a cutoff point by quicksort.

Chapter 8: Binary Trees

Overview

In this chapter we switch from algorithms, the focus of the last chapter on sorting, to data structures. Binary trees are one of the fundamental data storage structures used in programming. They provide advantages that the data structures we've seen so far cannot. In this chapter we'll learn why you would want to use trees, how they work, and how to go about creating them.

Why Use Binary Trees?

Why might you want to use a tree? Usually, because it combines the advantages of two other structures: an ordered array and a linked list. You can search a tree quickly, as you can an ordered array, and you can also insert and delete items quickly, as you can with a linked list. Let's explore these topics a bit before delving into the details of trees.

Slow Insertion in an Ordered Array

Imagine an array in which all the elements are arranged in order; that is, an ordered array, such as we saw in Chapter 3, "Simple Sorting." As we learned, it's quick to search such an array for a particular value, using a binary search. You check in the center of the array; if the object you're looking for is greater than what you find there, you narrow your search to the top half of the array; if it's less, you narrow your search to the bottom half. Applying this process repeatedly finds the object in O(logN) time. It's also quick to iterate through an ordered array, visiting each object in sorted order.

On the other hand, if you want to insert a new object into an ordered array, you first need to find where the object will go, and then move all the objects with greater keys up one space in the array to make room for it. These multiple moves are time consuming, requiring, on the average, moving half the items (N/2 moves). Deletion involves the same multimove operation, and is thus equally slow.

If you're going to be doing a lot of insertions and deletions, an ordered array is a bad choice.

Slow Searching in a Linked List

On the other hand, as we saw in Chapter 7, "Advanced Sorting," insertions and deletions are quick to perform on a linked list. They are accomplished simply by changing a few references. These operations require O(1) time (the fastest Big-O time).

Unfortunately, however, *finding* a specified element in a linked list is not so easy. You must start at the beginning of the list and visit each element until you find the one you're looking for. Thus you will need to visit an average of N/2 objects, comparing each one's key with the desired value. This is slow, requiring O(N) time. (Notice that times considered fast for a sort are slow for data structure operations.)

You might think you could speed things up by using an ordered linked list, in which the elements were arranged in order, but this doesn't help. You still must start at the beginning and visit the elements in order, because there's no way to access a given element without following the chain of references to it. (Of course, in an ordered list it's much quicker to visit the nodes in order than it is in a non-ordered list, but that doesn't

help to find an arbitrary object.)

Trees to the Rescue

It would be nice if there were a data structure with the quick insertion and deletion of a linked list, and also the quick searching of an ordered array. Trees provide both these characteristics, and are also one of the most interesting data structures.

What Is a Tree?

We'll be mostly interested in a particular kind of tree called a *binary tree*, but let's start by discussing trees in general before moving on to the specifics of binary trees.

A tree consists of *nodes* connected by *edges*. Figure 8.1 shows a tree. In such a picture of a tree (or in our Workshop applet) the nodes are represented as circles, and the edges as lines connecting the circles.

Trees have been studied extensively as abstract mathematical entities, so there's a large amount of theoretical knowledge about them. A tree is actually an instance of a more general category called a graph, but we don't need to worry about that here. We'll discuss graphs in Chapters 13, "Graphs," and 14, "Weighted Graphs."

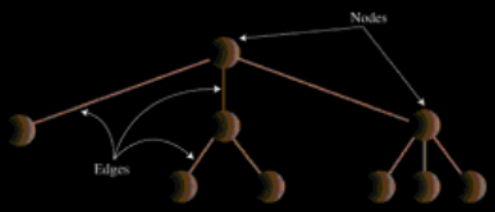

Figure 8.1: A tree

In computer programs, nodes often represent such entities as people, car parts, airline reservations, and so on; in other words, the typical items we store in any kind of data structure. In an OOP language such as Java, these real-world entities are represented by objects.

The lines (edges) between the nodes represent the way the nodes are related. Roughly speaking, the lines represent convenience: It's easy (and fast) for a program to get from one node to another if there is a line connecting them. In fact, the *only* way to get from node to node is to follow a path along the lines. Generally you are restricted to going in one direction along edges: from the root downward.

Edges are likely to be represented in a program by references, if the program is written in Java (or by pointers if the program is written in C or C++).

Typically there is one node in the top row of a tree, with lines connecting to more nodes on the second row, even more on the third, and so on. Thus trees are small on the top and large on the bottom. This may seem upside-down compared with real trees, but generally a program starts an operation at the small end of the tree, and it's (arguably) more natural to think about going from top to bottom, as in reading text.

There are different kinds of trees. The tree shown in Figure 8.1 has more than two children per node. (We'll see what "children" means in a moment.) However, in this chapter we'll be discussing a specialized form of tree called a *binary tree*. Each node in a binary tree has a maximum of two children. More general trees, in which nodes can have more than two children, are called multiway trees. We'll see an example in Chapter 10, "2-3-4 Tables and

External Storage," where we discuss 2-3-4 trees.

Why Use Binary Trees?

Why might you want to use a tree? Usually, because it combines the advantages of two other structures: an ordered array and a linked list. You can search a tree quickly, as you can an ordered array, and you can also insert and delete items quickly, as you can with a linked list. Let's explore these topics a bit before delving into the details of trees.

Slow Insertion in an Ordered Array

Imagine an array in which all the elements are arranged in order; that is, an ordered array, such as we saw in Chapter 3, "Simple Sorting." As we learned, it's quick to search such an array for a particular value, using a binary search. You check in the center of the array; if the object you're looking for is greater than what you find there, you narrow your search to the top half of the array; if it's less, you narrow your search to the bottom half. Applying this process repeatedly finds the object in O(logN) time. It's also quick to iterate through an ordered array, visiting each object in sorted order.

On the other hand, if you want to insert a new object into an ordered array, you first need to find where the object will go, and then move all the objects with greater keys up one space in the array to make room for it. These multiple moves are time consuming, requiring, on the average, moving half the items (N/2 moves). Deletion involves the same multimove operation, and is thus equally slow.

If you're going to be doing a lot of insertions and deletions, an ordered array is a bad choice.

Slow Searching in a Linked List

On the other hand, as we saw in Chapter 7, "Advanced Sorting," insertions and deletions are quick to perform on a linked list. They are accomplished simply by changing a few references. These operations require O(1) time (the fastest Big-O time).

Unfortunately, however, *finding* a specified element in a linked list is not so easy. You must start at the beginning of the list and visit each element until you find the one you're looking for. Thus you will need to visit an average of N/2 objects, comparing each one's key with the desired value. This is slow, requiring O(N) time. (Notice that times considered fast for a sort are slow for data structure operations.)

You might think you could speed things up by using an ordered linked list, in which the elements were arranged in order, but this doesn't help. You still must start at the beginning and visit the elements in order, because there's no way to access a given element without following the chain of references to it. (Of course, in an ordered list it's much quicker to visit the nodes in order than it is in a non-ordered list, but that doesn't help to find an arbitrary object.)

Trees to the Rescue

It would be nice if there were a data structure with the quick insertion and deletion of a linked list, and also the quick searching of an ordered array. Trees provide both these characteristics, and are also one of the most interesting data structures.

What Is a Tree?

We'll be mostly interested in a particular kind of tree called a *binary tree*, but let's start by discussing trees in general before moving on to the specifics of binary trees.

A tree consists of *nodes* connected by *edges*. Figure 8.1 shows a tree. In such a picture of a tree (or in our Workshop applet) the nodes are represented as circles, and the edges as lines connecting the circles.

Trees have been studied extensively as abstract mathematical entities, so there's a large amount of theoretical knowledge about them. A tree is actually an instance of a more general category called a graph, but we don't need to worry about that here. We'll discuss graphs in Chapters 13, "Graphs," and 14, "Weighted Graphs."

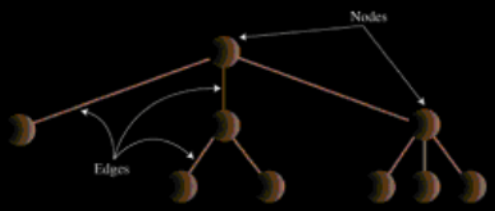

Figure 8.1: A tree

In computer programs, nodes often represent such entities as people, car parts, airline reservations, and so on; in other words, the typical items we store in any kind of data structure. In an OOP language such as Java, these real-world entities are represented by objects.

The lines (edges) between the nodes represent the way the nodes are related. Roughly speaking, the lines represent convenience: It's easy (and fast) for a program to get from one node to another if there is a line connecting them. In fact, the *only* way to get from node to node is to follow a path along the lines. Generally you are restricted to going in one direction along edges: from the root downward.

Edges are likely to be represented in a program by references, if the program is written in Java (or by pointers if the program is written in C or C++).

Typically there is one node in the top row of a tree, with lines connecting to more nodes on the second row, even more on the third, and so on. Thus trees are small on the top and large on the bottom. This may seem upside-down compared with real trees, but generally a program starts an operation at the small end of the tree, and it's (arguably) more natural to think about going from top to bottom, as in reading text.

There are different kinds of trees. The tree shown in Figure 8.1 has more than two children per node. (We'll see what "children" means in a moment.) However, in this chapter we'll be discussing a specialized form of tree called a *binary tree*. Each node in a binary tree has a maximum of two children. More general trees, in which nodes can have more than two children, are called multiway trees. We'll see an example in Chapter 10, "2-3-4 Tables and External Storage," where we discuss 2-3-4 trees.

An Analogy

One commonly encountered tree is the hierarchical file structure in a computer system. The root directory of a given device (designated with the backslash, as in `C:\`, on many systems) is the tree's root. The directories one level below the root directory are its children. There may be many levels of subdirectories. Files represent leaves; they have no children of their own.

Clearly a hierarchical file structure is not a binary tree, because a directory may have many children. A complete pathname, such as `C:\SALES\EAST\NOVEMBER\SMITH.DAT`, corresponds to the path from the root to the `SMITH.DAT` leaf. Terms used for the file structure, such as root and path, were borrowed

from tree theory.

A hierarchical file structure differs in a significant way from the trees we'll be discussing here. In the file structure, subdirectories contain no data; only references to other subdirectories or to files. Only files contain data. In a tree, every node contains data (a personnel record, car-part specifications, or whatever). In addition to the data, all nodes except leaves contain references to other nodes.

How Do Binary Trees Work?

Let's see how to carry out the common binary-tree operations of finding a node with a given key, inserting a new node, traversing the tree, and deleting a node. For each of these operations we'll first show how to use the Tree Workshop applet to carry it out; then we'll look at the corresponding Java code.

The Tree Workshop Applet

Start up the binary Tree Workshop applet. You'll see a screen something like that shown in Figure 8.5. However, because the tree in the Workshop applet is randomly generated, it won't look exactly the same as the tree in the figure.

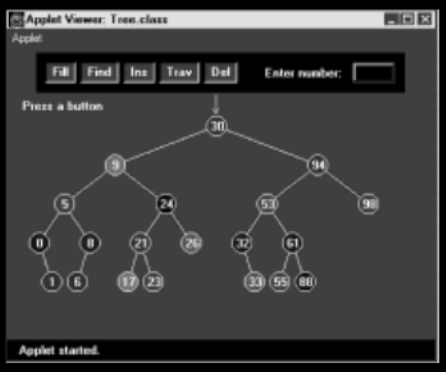

Figure 8.5: The binary Tree Workshop applet

Using the Applet

The key values shown in the nodes range from 0 to 99. Of course, in a real tree, there would probably be a larger range of key values. For example, if employees' Social Security numbers were used for key values, they would range up to 999,999,999.

Another difference between the Workshop applet and a real tree is that the Workshop applet is limited to a depth of 5; that is, there can be no more than 5 levels from the root to the bottom. This restriction ensures that all the nodes in the tree will be visible on the screen. In a real tree the number of levels is unlimited (until you run out of memory).

Using the Workshop applet, you can create a new tree whenever you want. To do this, click the Fill button. A prompt will ask you to enter the number of nodes in the tree. This can vary from 1 to 31, but 15 will give you a representative tree. After typing in the number, press Fill twice more to generate the new tree. You can experiment by creating trees with different numbers of nodes.

Unbalanced Trees

Notice that some of the trees you generate are *unbalanced*; that is, they have most of their nodes on one side of the root or the other, as shown in Figure 8.6. Individual

subtrees may also be unbalanced.

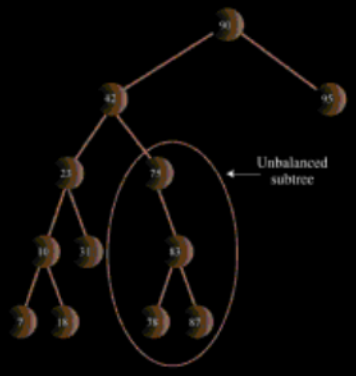

Figure 8.6: An unbalanced tree (with an unbalanced subtree)

Trees become unbalanced because of the order in which the data items are inserted. If these key values are inserted randomly, the tree will be more or less balanced. However, if an ascending sequence (like 11, 18, 33, 42, 65, and so on) or a descending sequence is generated, all the values will be right children (if ascending) or left children (if descending) and the tree will be unbalanced. The key values in the Workshop applet are generated randomly, but of course some short ascending or descending sequences will be created anyway, which will lead to local imbalances. When you learn how to insert items into the tree in the Workshop applet you can try building up a tree by inserting such an ordered sequence of items and see what happens.

If you ask for a large number of nodes when you use Fill to create a tree, you may not get as many nodes as you requested. Depending on how unbalanced the tree becomes, some branches may not be able to hold a full number of nodes. This is because the depth of the applet's tree is limited to five; the problem would not arise in a real tree.

If a tree is created by data items whose key values arrive in random order, the problem of unbalanced trees may not be too much of a problem for larger trees, because the chances of a long run of numbers in sequence is small. But key values can arrive in strict sequence; for example, when a data-entry person arranges a stack of personnel files into order of ascending employee number before entering the data. When this happens, tree efficiency can be seriously degraded. We'll discuss unbalanced trees and what to do about them in Chapter 9, "Red-Black Trees."

Representing the Tree in Java Code

Let's see how we might implement a binary tree in Java. As with other data structures, there are several approaches to representing a tree in the computer's memory. The most common is to store the nodes at unrelated locations in memory and connect them using references in each node that point to its children.

It's also possible to represent a tree in memory as an array, with nodes in specific positions stored in corresponding positions in the array. We'll return to this possibility at the end of this chapter. For our sample Java code we'll use the approach of connecting the nodes using references.

As we discuss individual operations we'll show code fragments pertaining to that operation. The complete program from which these fragments are extracted can be seen toward the end of this chapter in Listing 8.1.

The Node Class

First, we need a class of node objects. These objects contain the data representing the objects being stored (employees in an employee database, for example) and also references to each of the node's two children. Here's how that looks:

```
class Node
   {
   int iData;                    // data used as key value
   float fData;                  // other data
   node leftChild;               // this node's left child
   node rightChild;              // this node's right child

   public void displayNode()
      {
      // (see Listing 8.1 for method body)
      }

   }
```

Some programmers also include a reference to the node's parent. This simplifies some operations but complicates others, so we don't include it. We do include a method called `displayNode()` to display the node's data, but its code isn't relevant here.

There are other approaches to designing class `Node`. Instead of placing the data items directly into the node, you could use a reference to an object representing the data item:

```
class Node
   {
   person p1;                    // reference to person object
   node leftChild;               // this node's left child
   node rightChild;              // this node's right child

   }

class person
   {
   int iData;
   float fData;

   }
```

This makes it conceptually clearer that the node and the data item it holds aren't the same thing, but it results in somewhat more complicated code, so we'll stick to the first approach.

The `Tree` Class

We'll also need a class from which to instantiate the tree itself; the object that holds all the nodes. We'll call this class `Tree`. It has only one field: a `Node` variable that holds the root. It doesn't need fields for the other nodes because they are all accessed from the root.

The `Tree` class has a number of methods: some for finding, inserting, and deleting nodes, several for different kinds of traverses, and one to display the tree. Here's a skeleton version:

```
class Tree
   {
   private Node root;             // the only data field in Tree

   public void find(int key)
      {
      }
   public void insert(int id, double dd)
      {
      }
   public void delete(int id)
      {
      }
   // various other methods

   }  // end class Tree
```

The `TreeApp` Class

Finally, we need a way to perform operations on the tree. Here's how you might write a class with a `main()` routine to create a tree, insert three nodes into it, and then search for one of them. We'll call this class `TreeApp`:

```
class TreeApp
   {
   public static void main(String[] args)
      {
      Tree theTree = new Tree;        // make a tree

      theTree.insert(50, 1.5);        // insert 3 nodes
      theTree.insert(25, 1.7);
      theTree.insert(75, 1.9);

      node found = theTree.find(25);  // find node with key 25
      if(found != null)
         System.out.println("Found the node with key 25");
      else
         System.out.println("Could not find node with key 25");
      }  // end main()

   }  // end class TreeApp
```

In Listing 8.1 the `main()` routine provides a primitive user interface so you can decide from the keyboard whether you want to insert, find, delete, or perform other operations.

Next we'll look at individual tree operations: finding a node, inserting a node, traversing the tree, and deleting a node.

Finding a Node

Finding a node with a specific key is the simplest of the major tree operations, so let's start with that.

Remember that the nodes in a binary search tree correspond to objects containing information. They could be *person objects*, with an employee number as the key and also perhaps name, address, telephone number, salary, and other fields. Or they could represent car parts, with a part number as the key value and fields for quantity on hand, price, and so on. However, the only characteristics of each node that we can see in the Workshop applet are a number and a color. A node is created with these two characteristics and keeps them throughout its life.

Using the Workshop Applet to Find a Node

Look at the Workshop applet and pick a node, preferably one near the bottom of the tree (as far from the root as possible). The number shown in this node is its *key value*. We're going to demonstrate how the Workshop applet finds the node, given the key value.

For purposes of this discussion we'll assume you've decided to find the node representing the item with key value 57, as shown in Figure 8.7. Of course, when you run the Workshop applet you'll get a different tree and will need to pick a different key value.

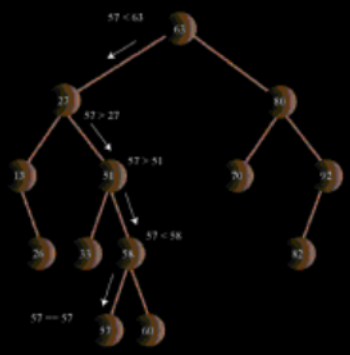

Figure 8.7: Finding node 57

Click the Find button. The prompt will ask for the value of the node to find. Enter 57 (or whatever the number is on the node you chose). Click Find twice more.

As the Workshop applet looks for the specified node, the prompt will display either "Going to left child" or "Going to right child," and the red arrow will move down one level to the right or left.

Figure 8.7 the arrow starts at the root. The program compares the key value 57 with the value at the root, which is 63. The key is less, so the program knows the desired node must be on the left side of the tree; either the root's left child or one of this child's descendants. The left child of the root has the value 27, so the comparison of 57 and 27 will show that the desired node is in the right subtree of 27. The arrow will go to 51, the root of this subtree. Here, 57 is again greater than the 51 node, so we go to the right, to 58, and then to the left, to 57. This time the comparison shows 57 equals the node's key value, so we've found the node we want.

The Workshop applet doesn't do anything with the node once it finds it, except to display a message saying it has been found. A serious program would perform some operation on the found node, such as displaying its contents or changing one of its fields.

Java Code for Finding a Node

Here's the code for the `find()` routine, which is a method of the `Tree` class:

```
public Node find(int key)       // find node with given key
   {                            // (assumes non-empty tree)
   Node current = root;                 // start at root

   while(current.iData != key)          // while no match,
      {
      if(key < current.iData)           // go left?
         current = current.leftChild;
      else
         current = current.rightChild; // or go right?
      if(current == null)               // if no child,
         return null;                   // didn't find it
      }
   return current;                      // found it

   }
```

This routine uses a variable `current` to hold the node it is currently examining. The argument `key` is the value to be found. The routine starts at the root. (It has to; this is the only node it can access directly.) That is, it sets `current` to the root.

Then, in the `while` loop, it compares the value to be found, `key`, with the value of the `iData` field (the key field) in the current node. If `key` is less than this field, then `current` is set to the node's left child. If `key` is greater than (or equal) to the node's `iData` field, then `current` is set to the node's right child.

Can't Find It

If `current` becomes equal to `null`, then we couldn't find the next child node in the sequence; we've reached the end of the line without finding the node we were looking for, so it can't exist. We return `null` to indicate this fact.

Found It

If the condition of the `while` loop is not satisfied, so that we exit from the bottom of the loop, then the `iData` field of `current` is equal to `key`; that is, we've found the node we want. We return the node, so that the routine that called `find()` can access any of the node's data.

Efficiency

As you can see, how long it takes to find a node depends on how many levels down it is situated. In the Workshop applet there can be up to 31 nodes, but no more than 5 levels—so you can find any node using a maximum of only 5 comparisons. This is O(logN) time, or more specifically $O(\log_2 N)$ time; the logarithm to the base 2. We'll discuss this further toward the end of this chapter.

Inserting a Node

To insert a node we must first find the place to insert it. This is much the same process as trying to find a node that turns out not to exist, as described in the section on Find. We follow the path from the root to the appropriate node, which will be the parent of the new node. Once this parent is found, the new node is connected as its left or right child, depending on whether the new node's key is less than or greater than that of the parent.

Using the Workshop Applet to Insert a Node

To insert a new node with the Workshop applet, press the Ins button. You'll be asked to type the key value of the node to be inserted. Let's assume we're going to insert a new node with the value 45. Type this into the text field.

The first step for the program in inserting a node is to find where it should be inserted. Figure 8.8a shows how this looks.

The value 45 is less than 60 but greater than 40, so we arrive at node 50. Now we want to go left because 45 is less than 50, but 50 has no left child; its `leftChild` field is null. When it sees this null, the insertion routine has found the place to attach the new node. The Workshop applet does this by creating a new node with the value 45 (and a randomly generated color) and connecting it as the left child of 50, as shown in Figure 8.8b.

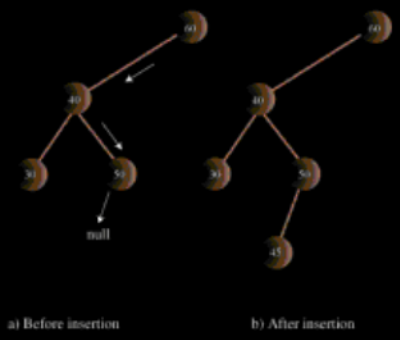

Figure 8.8: Inserting a node

Java Code for Inserting a Node

The `insert()` function starts by creating the new node, using the data supplied as arguments.

Next, `insert()` must determine where to insert the new node. This is done using roughly the same code as finding a node, described in the section on `find()`. The difference is that when you're simply trying to *find* a node and you encounter a null (non-existent) node, you know the node you're looking for doesn't exist so you return immediately. When you're trying to *insert* a node you insert it (creating it first, if necessary) before returning.

The value to be searched for is the data item passed in the argument `id`. The `while` loop uses `true` as its condition because it doesn't care if it encounters a node with the same value as `id`; it treats another node with the same key value as if it were simply greater than the key value. (We'll return to the subject of duplicate nodes later in this chapter.)

A place to insert a new node will always be found (unless you run out of memory); when it is, and the new node is attached, the `while` loop exits with a `return` statement.

Here's the code for the `insert()` function:

```
public void insert(int id, double dd)
```

```
        {
        Node newNode = new Node();     // make new node
        newNode.iData = id;            // insert data
        newNode.dData = dd;
        if(root==null)                 // no node in root
           root = newNode;
        else                           // root occupied
           {
           Node current = root;        // start at root
           Node parent;
           while(true)                 // (exits internally)
              {
              parent = current;
              if(id < current.iData)   // go left?
                 {
                 current = current.leftChild;
                 if(current == null)   // if end of the line,
                    {                  // insert on left
                    parent.leftChild = newNode;
                    return;
                    }
                 }  // end if go left
              else                     // or go right?
                 {
                 current = current.rightChild;
                 if(current == null)   // if end of the line
                    {                  // insert on right
                    parent.rightChild = newNode;
                    return;
                    }
                 }  // end else go right
              }  // end while
           }  // end else not root
        }  // end insert()

  // -------------------------------------------------------------
  -
```

We use a new variable, `parent` (the parent of `current`), to remember the last non-null node we encountered (50 in the figure). This is necessary because `current` is set to `null` in the process of discovering that its previous value did not have an appropriate child. If we didn't save `parent`, we'd lose track of where we were.

To insert the new node, change the appropriate child pointer in `parent` (the last non-null node you encountered) to point to the new node. If you were looking unsuccessfully for `parent`'s left child, you attach the new node as `parent`'s left child; if you were looking for its right child, you attach the new node as its right child. In Figure 8.8, 45 is attached as the left child of 50.

Traversing the Tree

Traversing a tree means visiting each node in a specified order. This process is not as commonly used as finding, inserting, and deleting nodes. One reason for this is that

traversal is not particularly fast. But traversing a tree is useful in some circumstances and the algorithm is interesting. (It's also simpler than deletion, the discussion of which we want to defer as long as possible.)

There are three simple ways to traverse a tree. They're called *preorder*, *inorder*, and *postorder*. The order most commonly used for binary search trees is inorder, so let's look at that first, and then return briefly to the other two.

Inorder Traversal

An inorder traversal of a binary search tree will cause all the nodes to be visited *in ascending order*, based on their key values. If you want to create a sorted list of the data in a binary tree, this is one way to do it.

The simplest way to carry out a traversal is the use of recursion (discussed in Chapter 6, "Recursion"). A recursive method to traverse the entire tree is called with a node as an argument. Initially, this node is the root. The method needs to do only three things:

1. Call itself to traverse the node's left subtree
2. Visit the node
3. Call itself to traverse the node's right subtree

Remember that *visiting* a node means doing something to it: displaying it, writing it to a file, or whatever.

Traversals work with any binary tree, not just with binary search trees. The traversal mechanism doesn't pay any attention to the key values of the nodes; it only concerns itself with whether a node has children.

Java Code for Traversing

The actual code for inorder traversal is so simple we show it before seeing how traversal looks in the Workshop applet. The routine, `inOrder()`, performs the three steps already described. The visit to the node consists of displaying the contents of the node. Like any recursive function, there must be a base case: the condition that causes the routine to return immediately, without calling itself. In `inOrder()` this happens when the node passed as an argument is null. Here's the code for the `inOrder()` method:

```
private void inOrder(node localRoot)
   {
   if(localRoot != null)
      {
      inOrder(localRoot.leftChild);
      localRoot.displayNode();
      inOrder(localRoot.rightChild);
      }

   }
```

This method is initially called with the root as an argument:

```
inOrder(root);
```

After that, it's on its own, calling itself recursively until there are no more nodes to visit.

Traversing a 3-Node Tree

Let's look at a simple example to get an idea of how this recursive traversal routine works. Imagine traversing a tree with only three nodes: a root (A) with a left child (B) and a right child (C), as shown in Figure 8.9.

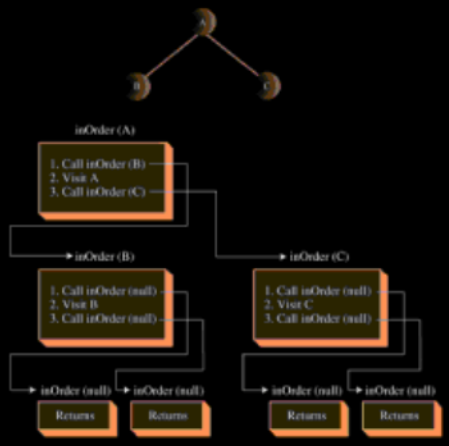

Figure 8.9: `inOrder()` method applied to 3-node tree

We start by calling `inOrder()` with the root A as an argument. This incarnation of `inOrder()` we'll call `inOrder(A)`. `inOrder(A)` first calls `inOrder()` with its left child, B, as an argument. This second incarnation of `inOrder()` we'll call `inOrder(B)`.

`inOrder(B)` now calls itself with its left child as an argument. However, it has no left child, so this argument is null. This creates an invocation of `inOrder()` we could call `inOrder(null)`. There are now three instances of `inOrder()` in existence: `inOrder(A)`, `inOrder(B)`, and `inOrder(null)`. However, `inOrder(null)` returns immediately when it finds its argument is null. (We all have days like that.)

Now `inOrder(B)` goes on to visit B; we'll assume this means to display it. Then `inOrder(B)` calls `inOrder()` again, with its right child as an argument. Again this argument is null, so the second `inOrder(null)` returns immediately. Now `inOrder(B)` has carried out steps 1, 2, and 3, so it returns (and thereby ceases to exist).

Now we're back to `inOrder(A)`, just returning from traversing A's left child. We visit A, and then call `inOrder()` again with C as an argument, creating `inOrder(C)`. Like `inOrder(B)`, `inOrder(C)` has no children, so step 1 returns with no action, step 2 visits C, and step 3 returns with no action. `inOrder(B)` now returns to `inOrder(A)`.

However, `inOrder(A)` is now done, so it returns and the entire traversal is complete. The order in which the nodes were visited is A, B, C; they have been visited *inorder*. In a binary search tree this would be the order of ascending keys.

More complex trees are handled similarly. The `inOrder()` function calls itself for each node, until it has worked its way through the entire tree.

Traversing with the Workshop Applet

To see what a traversal looks like with the Workshop applet, repeatedly press the Trav

button. (There's no need to type in any numbers.)

Here's what happens when you use the Tree Workshop applet to traverse inorder the tree shown in Figure 8.10. This is slightly more complex than the 3-node tree seen previously. The red arrow starts at the root. Table 8.1 shows the sequence of node keys and the corresponding messages. The key sequence is displayed at the bottom of the Workshop applet screen.

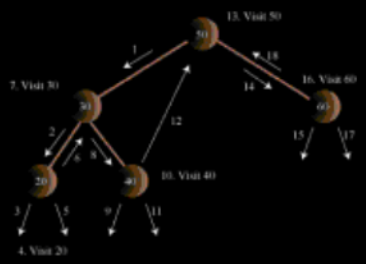

Figure 8.10: Traversing a tree inorder

Table 8.1: WORKSHOP APPLET TRAVERSAL

Step Number	Red Arrow on Node	Message	List of Nodes Visited
1	50 (root)	Will check left child	
2	30	Will check left child	
3	20	Will check left child	
4	20	Will visit this node	
5	20	Will check right child	20
6	20	Will go to root of previous subtree	20
7	30	Will visit this node	20
8	30	Will check for right child	20 30
9	40	Will check left child	20 30
10	40	Will visit this node	20 30
11	40	Will check right child	20 30 40
12	40	Will go to root of previous subtree	20 30 40

13	50	Will visit this node	20 30 40
14	50	Will check right child	20 30 40 50
15	60	Will check left child	20 30 40 50
16	60	Will visit this node	20 30 40 50
17	60	Will check for right child	20 30 40 50 60
18	60	Will go to root of previous subtree	20 30 40 50 60
19	50	Done traversal	20 30 40 50 60

It may not be obvious, but for each node, the routine traverses the node's left subtree, visits the node, and traverses the right subtree. For example, for node 30 this happens in steps 2, 7, and 8.

All this isn't as complicated as it looks. The best way to get a feel for what's happening is to traverse a variety of different trees with the Workshop applet.

Preorder and Postorder Traversals

You can traverse the tree in two ways besides inorder; they're called preorder and post-order. It's fairly clear why you might want to traverse a tree inorder, but the motivation for preorder and postorder traversals is more obscure. However, these traversals are indeed useful if you're writing programs that *parse* or analyze algebraic expressions. Let's see why that should be true.

A binary tree (not a binary search tree) can be used to represent an algebraic expression that involves the binary arithmetic operators `+`, `-`, `/`, and `*`. The root node holds an operator, and each of its subtrees represents either a variable name (like A, B, or C) or another expression.

For example, the binary tree shown in Figure 8.11 represents the algebraic expression

```
A*(B+C)
```

This is called *infix* notation; it's the notation normally used in algebra. Traversing the tree inorder will generate the correct inorder sequence A*B+C, but you'll need to insert the parentheses yourself.

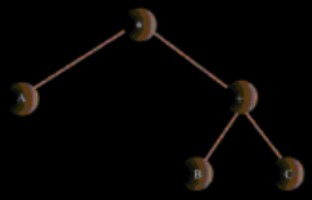

Figure 8.11: Tree representing an algebraic expression

What's all this got to do with preorder and postorder traversals? Let's see what's involved. For these other traversals the same three steps are used as for inorder, but in a different sequence. Here's the sequence for a `preorder()` method:

1. Visit the node.

2. Call itself to traverse the node's left subtree.

3. Call itself to traverse the node's right subtree.

Traversing the tree shown in Figure 8.11 using preorder would generate the expression

```
*A+BC
```

This is called *prefix* notation. One of the nice things about it is that parentheses are never required; the expression is unambiguous without them. It means "apply the operator * to the next two things in the expression." These two things are A and +BC. The expression +BC means "apply + to the next two things in the expression;" which are B and C, so this last expression is B+C in inorder notation. Inserting that into the original expression *A+BC (preorder) gives us A*(B+C) in inorder.

The postorder traversal method contains the three steps arranged in yet another way:

1. Call itself to traverse the node's left subtree.

2. Call itself to traverse the node's right subtree.

3. Visit the node.

For the tree in Figure 8.11, visiting the nodes with a postorder traversal would generate the expression

```
ABC+*
```

This is called *postfix* notation. As described in Chapter 4, "Stacks and Queues," it means "apply the last operator in the expression, `*`, to the first and second things." The first thing is A, and the second thing is BC+.

BC+ means "apply the last operator in the expression, `+`, to the first and second things." The first thing is B and the second thing is C, so this gives us (B+C) in infix. Inserting this in the original expression ABC+* (postfix) gives us A*(B+C) postfix.

The code in Listing 8.1 contains methods for preorder and postorder traversals, as well as for inorder.

Finding Maximum and Minimum Values

Incidentally, we should note how easy it is to find the maximum and minimum values in a binary search tree. In fact, it's so easy we don't include it as an option in the Workshop applet, nor show code for it in Listing 8.1. Still, it's important to understand how it works.

For the minimum, go to the left child of the root; then go to the left child of that child, and

so on, until you come to a node that has no left child. This node is the minimum, as shown in Figure 8.12.

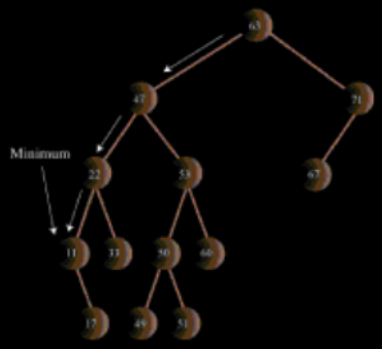

Figure 8.12: Minimum value of a tree

Here's some code that returns the node with the minimum key value:

```
public Node minimum()      // returns node with minimum key
value
   {
   Node current, last;
   current = root;                  // start at root
   while(current != null)           // until the bottom,
      {
      last = current;               // remember node
      current = current.leftChild;  // go to left child
      }
   return last;

   }
```

We'll need to know about finding the minimum value when we set about deleting a node.

For the *maximum* value in the tree, follow the same procedure but go from right child to right child until you find a node with no right child. This node is the maximum. The code is the same except that the last statement in the loop is

```
current = current.rightChild;  // go to right child
```

Deleting a Node

Deleting a node is the most complicated common operation required for binary search trees. However, deletion is important in many tree applications, and studying the details builds character.

You start by finding the node you want to delete, using the same approach we saw in `find()` and `insert()`. Once you've found the node, there are three cases to consider.

1. The node to be deleted is a leaf (has no children).

2. The node to be deleted has one child.

3. The node to be deleted has two children.

We'll look at these three cases in turn. The first is easy, the second almost as easy, and the third quite complicated.

Case 1: The Node to be Deleted Has No Children

To delete a leaf node, you simply change the appropriate child field in the node's parent to point to null instead of to the node. The node will still exist, but it will no longer be part of the tree. This is shown in Figure 8.13.

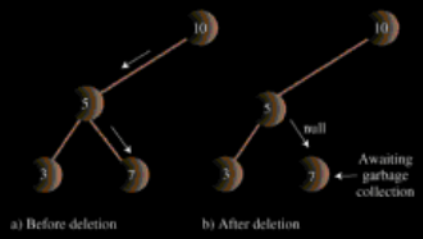

Figure 8.13: Deleting a node with no children

Because of Java's garbage collection feature, we don't need to worry about explicitly deleting the node itself. When Java realizes that nothing in the program refers to the node, it will be removed from memory. (In C and C++ you would need to execute `free()` or `delete()` to remove the node from memory.)

Using the Workshop Applet to Delete a Node With No Children

Assume you're going do delete node 7 in Figure 8.13. Press the Del button and enter 7 when prompted. Again, the node must be found before it can be deleted. Repeatedly pressing Del will take you from 10 to 5 to 7. When it's found, it's deleted without incident.

Java Code to Delete a Node With No Children

The first part of the `delete()` routine is similar to `find()` and `insert()`. It involves finding the node to be deleted. As with `insert()`, we need to remember the parent of the node to be deleted so we can modify its child fields. If we find the node, we drop out of the `while` loop with `parent` containing the node to be deleted. If we can't find it, we return from `delete()` with a value of `false`.

```
public boolean delete(int key) // delete node with given key
   {                           // (assumes non-empty list)
   Node current = root;
   Node parent = root;
   boolean isLeftChild = true;

   while(current.iData != key)        // search for node
      {
      parent = current;
      if(key < current.iData)         // go left?
         {
         isLeftChild = true;
         current = current.leftChild;
         }
      else                            // or go right?
```

```
            {
            isLeftChild = false;
            current = current.rightChild;
            }
         if(current == null)              // end of the line,
            return false;                 // didn't find it
         }  // end while
      // found node to delete
      // continues...

      }
```

Once we've found the node, we check first to see whether it has no children. When this is true we check the special case of the root; if that's the node to be deleted, we simply set it to null, and this empties the tree. Otherwise, we set the parent's `leftChild` or `rightChild` field to null to disconnect the parent from the node.

```
   // delete() continued...
   // if no children, simply delete it
   if(current.leftChild==null &&
                              current.rightChild==null)
      {
      if(current == root)             // if root,
         root = null;                 // tree is empty
      else if(isLeftChild)
         parent.leftChild = null;     // disconnect
      else                            // from parent
         parent.rightChild = null;
      }

   // continues...
```

Case 2: The Node to be Deleted Has One Child

This case isn't so bad either. The node has only two connections: to its parent and to its only child. You want to "snip" the node out of this sequence by connecting its parent directly to its child. This involves changing the appropriate reference in the parent (`leftChild` or `rightChild`) to point to the deleted node's child. This is shown in Figure 8.14.

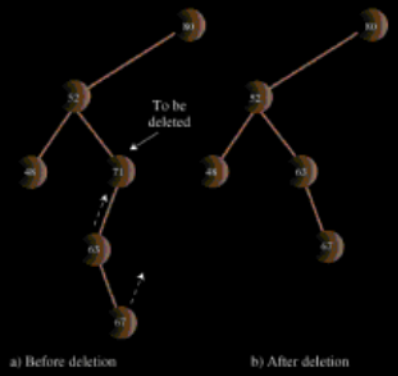

Figure 8.14: Deleting a node with one child

Using the Workshop Applet to Delete a Node with One Child

Let's assume we're using the Workshop on the tree in Figure 8.14, and deleting node 71, which has a left child but no right child. Press Del and enter 71 when prompted. Keep pressing Del until the arrow rests on 71. Node 71 has only one child, 63. It doesn't matter whether 63 has children of its own; in this case, it has one: 67.

Pressing Del once more causes 71 to be deleted. Its place is taken by its left child, 63. In fact, the entire subtree of which 63 is the root is moved up and plugged in as the new right child of 52.

Use the Workshop applet to generate new trees with one-child nodes, and see what happens when you delete them. Look for the subtree whose root is the deleted node's child. No matter how complicated this subtree is, it's simply moved up and plugged in as the new child of the deleted node's parent.

Java Code to Delete a Node With One Child

The following code shows how to deal with the one-child situation. There are four variations: the child of the node to be deleted may be either a left or right child, and for each of these cases the node to be deleted may be either the left or right child of its parent.

There is also a specialized situation: The node to be deleted may be the root, in which case it has no parent and is simply replaced by the appropriate subtree. Here's the code (which continues from the end of the no-child code fragment shown earlier):

```
// delete() continued...
// if no right child, replace with left subtree
else if(current.rightChild==null)
   if(current == root)
      root = current.leftChild;
   else if(isLeftChild)           // left child of parent
      parent.leftChild = current.leftChild;
   else                           // right child of parent
      parent.rightChild = current.leftChild;

// if no left child, replace with right subtree
else if(current.leftChild==null)
   if(current == root)
      root = current.rightChild;
   else if(isLeftChild)           // left child of parent
      parent.leftChild = current.rightChild;
   else                           // right child of parent
      parent.rightChild = current.rightChild;

// continued...
```

Notice that working with references makes it easy to move an entire subtree. You do this by simply disconnecting the old reference to the subtree and creating a new reference to it somewhere else. Although there may be lots of nodes in the subtree, youdon't need to worry about moving them individually. In fact, they only "move" in the sense of being conceptually in different positions relative to the other nodes. As far as the program is concerned, only the reference to the root of the subtree has changed.

Case 3: The Node to be Deleted Has Two Children

Now the fun begins. If the deleted node has two children, you can't just replace it with one of these children, at least if the child has its own children. Why not? Examine Figure 8.15, and imagine deleting node 25 and replacing it with its right subtree, whose root is 35. Which left child would 35 have? The deleted node's left child, 15, or the new node's left child, 30? In either case 30 would be in the wrong place, but we can't just throw it away.

We need another approach. The good news is that there's a trick. The bad news is that, even with the trick, there are a lot of special cases to consider. Remember that in a binary search tree the nodes are arranged in order of ascending keys. For each node, the node with the next-highest key is called its *inorder successor*, or simply its successor. In Figure 8.15a, node 30 is the successor of node 25.

Here's the trick: To delete a node with two children, *replace the node with its inorder successor*. Figure 8.16 shows a deleted node being replaced by its successor. Notice that the nodes are still in order. (There's more to it if the successor itself has children; we'll look at that possibility in a moment.)

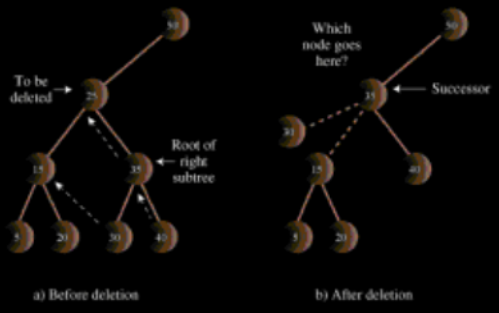

Figure 8.15: Can't replace with subtree

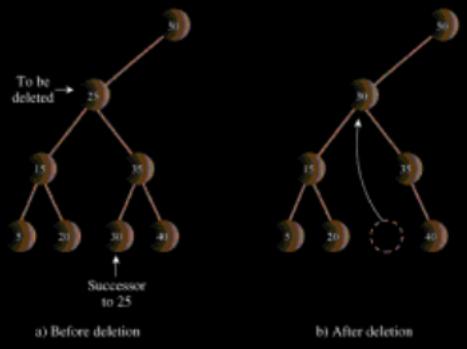

Figure 8.16: Node replaced by its successor

Finding the Successor

How do you find the successor of a node? As a human being, you can do this quickly (for small trees, anyway). Just take a quick glance at the tree and find the next-largest number following the key of the node to be deleted. In Figure 8.16 it doesn't take long to see that the successor of 25 is 30. There's just no other number which is greater than 25 and also smaller than 35. However, the computer can't do things "at a glance," it needs an algorithm.

First the program goes to the original node's right child, which must have a key larger than the node. Then it goes to this right child's left child (if it has one), and to this left child's left child, and so on, following down the path of left children. The last left child in this path is the successor of the original node, as shown in Figure 8.17.

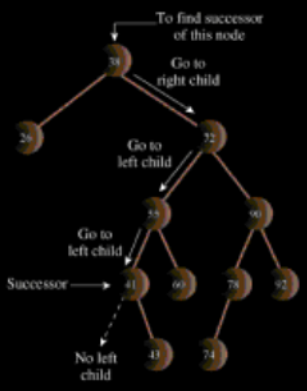

Figure 8.17: Finding the successor

Why does this work? What we're really looking for is *the smallest of the set of nodes that are larger than the original node*. When you go to the original node's right child, all the nodes in the resulting subtree are greater than the original node, because this is how a binary search tree is defined. Now we want the smallest value in this subtree. As we learned, you can find the minimum value in a subtree by following the path down all the left children. Thus, this algorithm finds the minimum value that is greater than the original node; this is what we mean by its successor.

If the right child of the original node has no left children, then this right child is itself the successor, as shown in Figure 8.18.

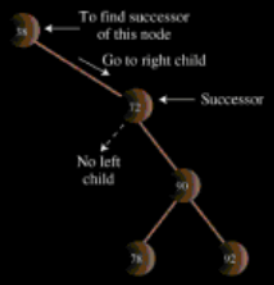

Figure 8.18: The right child is the successor

Using the Workshop Applet to Delete a Node with Two Children

Generate a tree with the Workshop applet, and pick a node with two children. Now mentally figure out which node is its successor by going to its right child and then following down the line of this right child's left children (if it has any). You may want to make sure the successor has no children of its own. If it does, the situation gets more complicated because entire subtrees are moved around, rather than a single node.

Once you've chosen a node to delete, click the Del button. You'll be asked for the key value of the node to delete. When you've specified it, repeated presses of the Del button will show the red arrow searching down the tree to the designated node. When the node is deleted, it's replaced by its successor.

In the example shown in Figure 8.15, the red arrow will go from the root at 50 to 25; then 25 will be replaced by 30.

Java Code to Find the Successor

Here's some code for a method `getSuccessor()`, which returns the successor of the node specified as its `delNode` argument. (This routine assumes that `delNode` does indeed have a right child, but we know this is true because we've already determined that the node to be deleted has two children.)

```
// returns node with next-highest value after delNode
// goes to right child, then right child's left descendants

private node getSuccessor(node delNode)

   {
   Node successorParent = delNode;
   Node successor = delNode;
   Node current = delNode.rightChild;   // go to right child
   while(current != null)               // until no more
      {                                 // left children,
      successorParent = successor;
      successor = current;
      current = current.leftChild;      // go to left child
      }
                                        // if successor not

   if(successor != delNode.rightChild)  // right child,
      {                                 // make connections
      successorParent.leftChild = successor.rightChild;
      successor.rightChild = delNode.rightChild;
      }

   return successor;

   }
```

The routine first goes to `delNode`'s right child, then, in the `while` loop, follows down the path of all this right child's left children. When the `while` loop exits, `successor` contains `delNode`'s successor.

Once we've found the successor, we may need to access its parent, so within the `while` loop we also keep track of the parent of the current node.

The `getSuccessor()` routine carries out two additional operations in addition to finding the successor. However, to understand these, we need to step back and consider the big picture.

As we've seen, the successor node can occupy one of two possible positions relative to `current`, the node to be deleted. The successor can be `current`'s right child, or it can be one of this right child's left descendants. We'll look at these two situations in turn.

`successor` Is Right Child of `delNode`

If `successor` is the right child of `delNode`, things are simplified somewhat because we

can simply move the subtree of which successor is the root and plug it in where the deleted node was. This requires only two steps:

1. Unplug `current` from the `rightChild` field of its parent (or `leftChild` field if appropriate), and set this field to point to `successor`.

2. Unplug current's left child from current, and plug it into the leftChild field of successor.

Here are the code statements that carry out these steps, excerpted from `delete()`:

1. `parent.rightChild = successor;`

2. successor.leftChild = current.leftChild;

This situation is summarized in Figure 8.19, which shows the connections affected by these two steps.

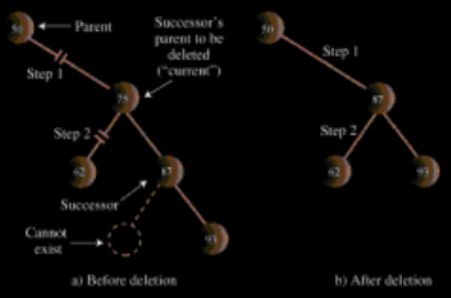

Figure 8.19: Deletion when successor is right child

Here's the code in context (a continuation of the `else-if` ladder shown earlier):

```
      // delete() continued

      else  // two children, so replace with inorder successor
         {
         // get successor of node to delete (current)
         Node successor = getSuccessor(current);

         // connect parent of current to successor instead
         if(current == root)
            root = successor;
         else if(isLeftChild)
            parent.leftChild = successor;
         else
            parent.rightChild = successor;
         // connect successor to current's left child
         successor.leftChild = current.leftChild;
         }  // end else two children

      // (successor cannot have a left child)
   return true;

   }  // end delete()
```

Notice that this is—finally—the end of the `delete()` routine. Let's review the code for these two steps.

Step 1: If the node to be deleted, `current`, is the root, it has no parent so we merely set the root to the successor. Otherwise, the node to be deleted can be either a left or right child (the figure shows it as a right child), so we set the appropriate field in its parent to point to `successor`. Once `delete()` returns and `current` goes out of scope, the node referred to by `current` will have no references to it, so it will be discarded during Java's next garbage collection.

Step 2: We set the left child of `successor` to point to `current`'s left child.

What happens if the successor has children of its own? First of all, *a successor node is guaranteed not to have a left child.* This is true whether the successor is the right child of the node to be deleted or one of this right child's left children. How do we know this?

Well, remember that the algorithm we use to determine the successor goes to the right child first, and then to any left children of that right child. It stops when it gets to a node with no left child, so the algorithm itself determines that the successor can't have any left children. If it did, that left child would be the successor instead.

You can check this out on the Workshop applet. No matter how many trees you make, you'll never find a situation in which a node's successor has a left child (assuming the original node has two children, which is the situation that leads to all this trouble in the first place).

On the other hand, the successor may very well have a right child. This isn't much of a problem when the successor is the right child of the node to be deleted. When we move the successor, its right subtree simply follows along with it. There's no conflict with the right child of the node being deleted, because the successor *is* this right child.

In the next section we'll see that a successor's right child needs more attention if the successor is not the right child of the node to be deleted.

`successor` Is Left Descendant of Right Child of `delNode`

If `successor` is a left descendant of the right child of the node to be deleted, four steps are required to perform the deletion:

1. Plug the right child of `successor` into the `leftChild` field of the successor's parent.

2. Plug the right child of the node to be deleted into the `rightChild` field of `successor`.

3. Unplug `current` from the `rightChild` field of its parent, and set this field to point to `successor`.

4. Unplug `current`'s left child from `current`, and plug it into the `leftChild` field of successor.

Steps 1 and 2 are handled in the `getSuccessor()` routine, while 3 and 4 are carried in `delete()`. Figure 8.20 shows the connections affected by these four steps.

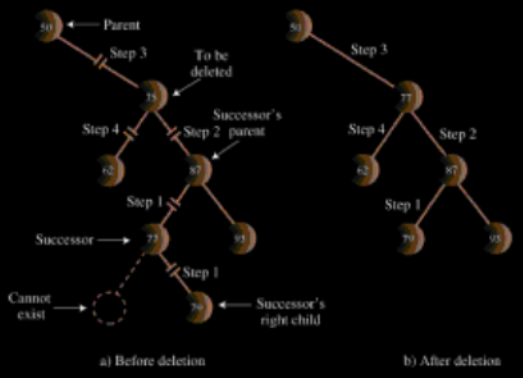

Figure 8.20: Deletion when successor is left child

Here's the code for these four steps:

1. `successorParent.leftChild = successor.rightChild;`

2. `successor.rightChild = delNode.rightChild;`

3. `parent.rightChild = successor;`

4. successor.leftChild = current.leftChild;

(Step 3 could also refer to the left child of its parent.) The numbers in Figure 8.20 show the connections affected by the four steps. Step 1 in effect *replaces the successor with its right subtree*. Step 2 keeps the right child of the deleted node in its proper place (this happens automatically when the successor is the right child of the deleted node). Steps 1 and 2 are carried out in the `if` statement that ends the `getSuccessor()` method shown earlier. Here's that statement again:

```
                                    // if successor not

   if(successor != delNode.rightChild)  // right child,
      {                                 // make connections
      successorParent.leftChild = successor.rightChild;
      successor.rightChild = delNode.rightChild;

      }
```

These steps are more convenient to perform here than in `delete()`, because in `getSuccessor()` it's easy to figure out where the successor's parent is while we're descending the tree to find the successor.

Steps 3 and 4 we've seen already; they're the same as steps 1 and 2 in the case where the successor is the right child of the node to be deleted, and the code is in the `if` statement at the end of `delete()`.

Is Deletion Necessary?

If you've come this far, you can see that deletion is fairly involved. In fact, it's so complicated that some programmers try to sidestep it altogether. They add a new Boolean field to the `node` class, called something like `isDeleted`. To delete a node, they simply set this field to `true`. Then other operations, like `find()`, check this field to

be sure the node isn't marked as deleted before working with it. This way, deleting a node doesn't change the structure of the tree. Of course, it also means that memory can fill up with "deleted" nodes.

This approach is a bit of a cop-out, but it may be appropriate where there won't be many deletions in a tree. (If ex-employees remain in the personnel file forever, for example.)

The Efficiency of Binary Trees

As you've seen, most operations with trees involve descending the tree from level to level to find a particular node. How long does it take to do this? In a full tree, about half the nodes are on the bottom level. (Actually there's one more node on the bottom row than in the rest of the tree.) Thus about half of all searches or insertions or deletions require finding a node on the lowest level. (An additional quarter of these operations require finding the node on the next-to-lowest level, and so on.)

During a search we need to visit one node on each level so we can get a good idea how long it takes to carry out these operations by knowing how many levels there are. Assuming a full tree, Table 8.2 shows how many levels are necessary to hold a given number of nodes.

Table 8.2: NUMBER OF LEVELS FOR SPECIFIED NUMBER OF NODES

Number of Nodes	Number of Levels
1	1
3	2
7	3
15	4
31	5
...	...
1,023	10
...	...
32,767	15
...	...
1,048,575	20
...	...
33,554,432	25

...	...
1,073,741,824	30

This situation is very much like the ordered array discussed in Chapter 3. In that case, the number of comparisons for a binary search was approximately equal to the base-2 logarithm of the number of cells in the array. Here, if we call the number of nodes in the first column N, and the number of levels in the second column L, then we can say that N is 1 less than 2 raised to the power L, or

$$N = 2^L - 1$$

Adding 1 to both sides of the equation, we have

$$N+1 = 2^L$$

This is equivalent to

$$L = \log_2(N+1)$$

Thus the time needed to carry out the common tree operations is proportional to the base-2 log of N. In Big-O notation we say such operations take O(logN) time.

If the tree isn't full, analysis is difficult. We can say that for a tree with a given number of levels, average search times will be shorter for the non-full tree than the full tree because fewer searches will proceed to lower levels.

Compare the tree to the other data-storage structures we've discussed so far. In an unordered array or a linked list containing 1,000,000 items, it would take you on the average 500,000 comparisons to find the one you wanted. But in a tree of 1,000,000 items, it takes 20 (or fewer) comparisons.

In an ordered array you can find an item equally quickly, but inserting an item requires, on the average, moving 500,000 items. Inserting an item in a tree with 1,000,000 items requires 20 or fewer comparisons, plus a small amount of time to connect the item.

Similarly, deleting an item from a 1,000,000-item array requires moving an average of 500,000 items, while deleting an item from a 1,000,000-node tree requires 20 or fewer comparisons to find the item, plus (possibly) a few more comparisons to find its successor, plus a short time to disconnect the item and connect its successor.

Thus a tree provides high efficiency for all the common data-storage operations.

Traversing is not as fast as the other operations. However, traversals are probably not very commonly carried out in a typical large database. They're more appropriate when a tree is used as an aid to parsing algebraic or similar expressions, which are probably not too long anyway.

Trees Represented as Arrays

Our code examples are based on the idea that a tree's edges are represented by `leftChild` and `rightChild` references in each node. However, there's a completely different way to represent a tree: with an array.

In the array approach, the nodes are stored in an array and are not linked by references.

The position of the node in the array corresponds to its position in the tree. The node at index 0 is the root, the node at index 1 is the root's left child, and so on, progressing from left to right along each level of the tree. This is shown in Figure 8.21.

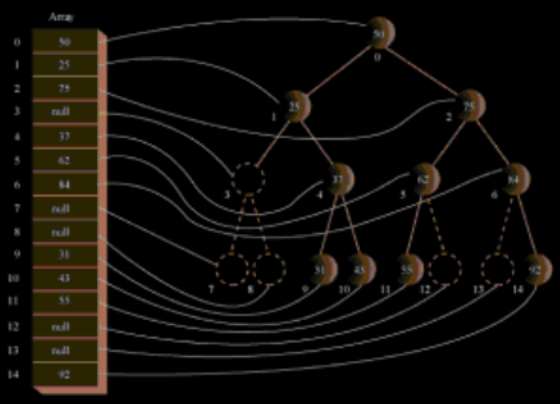

Figure 8.21: Tree represented by an array

Every position in the tree, whether it represents an existing node or not, corresponds to a cell in the array. Adding a node at a given position in the tree means inserting the node into the equivalent cell in the array. Cells representing tree positions with no nodes are filled with zero or `null`.

With this scheme, a node's children and parent can be found by applying some simple arithmetic to the node's index number in the array. If a node's index number is `index`, then this node's left child is

```
2*index + 1
```

its right child is

```
2*index + 2
```

and its parent is

```
(index-1) / 2
```

(where the '`/`' character indicates integer division with no remainder). You can check this out by looking at the figure.

In most situations, representing a tree with an array isn't very efficient. Unfilled nodes and deleted nodes leave holes in the array, wasting memory. Even worse, when deletion of a node involves moving subtrees, every node in the subtree must be moved to its new location in the array, which is time-consuming in large trees.

However, if deletions aren't allowed, then the array representation may be useful, especially if obtaining memory for each node dynamically is, for some reason, too time-consuming. The array representation may also be useful in special situations. The tree in the Workshop applet, for example, is represented internally as an array to make it easy to map the nodes from the array to fixed locations on the screen display.

Duplicate Keys

As in other data structures, the problem of duplicate keys must be addressed. In the code

shown for `insert()`, and in the Workshop applet, a node with a duplicate key will be inserted as the right child of its twin.

The problem is that the `find()` routine will find only the first of two (or more) duplicate nodes. The `find()` routine could be modified to check an additional data item, to distinguish data items even when the keys were the same, but this would be (at least somewhat) time-consuming.

One option is to simply forbid duplicate keys. When duplicate keys are excluded by the nature of the data (employee ID numbers, for example) there's no problem. Otherwise, you need to modify the `insert()` routine to check for equality during the insertion process, and abort the insertion if a duplicate is found.

The Fill routine in the Workshop applet excludes duplicates when generating the random keys.

Duplicate Keys

As in other data structures, the problem of duplicate keys must be addressed. In the code shown for `insert()`, and in the Workshop applet, a node with a duplicate key will be inserted as the right child of its twin.

The problem is that the `find()` routine will find only the first of two (or more) duplicate nodes. The `find()` routine could be modified to check an additional data item, to distinguish data items even when the keys were the same, but this would be (at least somewhat) time-consuming.

One option is to simply forbid duplicate keys. When duplicate keys are excluded by the nature of the data (employee ID numbers, for example) there's no problem. Otherwise, you need to modify the `insert()` routine to check for equality during the insertion process, and abort the insertion if a duplicate is found.

The Fill routine in the Workshop applet excludes duplicates when generating the random keys.

Summary

- Trees consist of nodes (circles) connected by edges (lines).
- The root is the topmost node in a tree; it has no parent.
- In a binary tree, a node has at most two children.
- In a binary search tree, all the nodes that are left descendants of node A have key values less than A; all the nodes that are A's right descendants have key values greater than (or equal to) A.
- Trees perform searches, insertions, and deletions in O(log N) time.
- Nodes represent the data-objects being stored in the tree.
- Edges are most commonly represented in a program by references to a node's children (and sometimes to its parent).
- Traversing a tree means visiting all its nodes in some order.

- The simplest traversals are preorder, inorder, and postorder.
- An unbalanced tree is one whose root has many more left descendents than right descendants, or vice versa.
- Searching for a node involves comparing the value to be found with the key value of a node, and going to that node's left child if the key search value is less, or to the node's right child if the search value is greater.
- Insertion involves finding the place to insert the new node, and then changing a child field in its new parent to refer to it.
- An inorder traversal visits nodes in order of ascending keys.
- Preorder and postorder traversals are useful for parsing algebraic expressions.
- When a node has no children, it can be deleted by setting the child field in its parent to null.
- When a node has one child, it can be deleted by setting the child field in its parent to point to its child.
- When a node has two children, it can be deleted by replacing it with its successor.
- The successor to a node A can be found by finding the minimum node in the subtree whose root is A's right child.
- In a deletion of a node with two children, different situations arise, depending on whether the successor is the right child of the node to be deleted or one of the right child's left descendants.
- Nodes with duplicate key values may cause trouble in arrays because only the first one can be found in a search.
- Trees can be represented in the computer's memory as an array, although the reference-based approach is more common.

Chapter 9: Red-Black Trees

Overview

As you learned in the last chapter, ordinary binary search trees offer important advantages as data storage devices: You can quickly search for an item with a given key, and you can also quickly insert or delete an item. Other data storage structures, such as arrays, sorted arrays, and linked lists, perform one or the other of these activities slowly. Thus binary search trees might appear to be the ideal data storage structure.

Unfortunately, ordinary binary search trees suffer from a troublesome problem. They work well if the data is inserted into the tree in random order. However, they become much slower if data is inserted in already sorted order (17, 21, 28, 36,…) or inversely sorted order (36, 28, 21, 17,…). When the values to be inserted are already ordered, a binary tree becomes unbalanced. With an unbalanced tree, the capability to quickly find (or insert or delete) a given element is lost.

This chapter explores one way to solve the problem of unbalanced trees: the red-black tree, which is a binary search tree with some added features.